Fermilab FERMILAB-Conf-00/083 February 2001

Farm Batch System and Fermi Inter-Process Communication and Synchronization Toolkit

M. Breitung, J. Fromm, T. Levshina, I. Mandrichenko, M. Schweitzer

Fermi National Accelerator Laboratory,
Batavia, Illinois, U.S.A.

Abstract

Farms Batch System (FBS) was developed as a batch process management system for off-line Run II data processing at Fermilab. FBS will manage PC farms composed of up to 250 nodes and scalable to 1000 nodes with disk capacity of up to several TB. FBS allows users to start arrays of parallel processes on multiple computers. It uses a simplified "resource counting" method load balancing. FBS has been successfully used for more than a year at Fermilab by fixed target experiments and will be used for collider experiment off-line data processing.

Fermi Inter-Process Communication toolkit (FIPC) was designed as a supplement product for FBS that helps establish synchronization and communication between processes running in a distributed batch environment. However, FIPC is an independent package, and can be used with other batch systems, as well as in a non-batch environment. FIPC provides users with a variety of global distributed objects such as semaphores, queues and string variables. Other types of objects can be easily added to FIPC.

FIPC has been running on several PC farms at Fermilab for half a year and is going to be used by CDF for off-line data processing.

Keywords: PC farms, batch system, computational resource management

1 Introduction

Large farms of inexpensive Intel-based computers running Linux OS are going to play increasingly important role as main tool for Run II off-line data processing at Fermilab. Computer power needs for Run II experiments are estimated to reach about 350 KMIPS by year 2001. Estimated size of PC farm used by a Run II experiment is about 250 dual-CPU 500 MHz Pentium computers. There will be more than 500 processes concurrently running on the farm, 2-3 processes per computer. In order to operate such a farm, it is necessary to have production management system responsible for allocation of computational resources as well as user process scheduling, monitoring and control. Due to large number of simultaneously running farm processes, in order to simplify process management and bookkeeping, it seems to be necessary to be able to group concurrent processes into "parallel jobs", and use such parallel job, instead of individual process, as the unit of operation of the production management system.

Some popular batch systems such as commercial LSF [2] and PBS [1] (Portable Batch System) were explored and evaluated. As a result, it was found that these systems do not match all of the requirements for farm-based off-line data processing production management system.

Farm Batch System (FBS) [3] was proposed, designed and developed to be a batch process management and resource utilization control system that will be used for farm data processing.

2 Farm Batch System (FBS)

2.1 Requirements

FBS design scalability requirements are more ambitious than those dictated by anticipated size of a typical Run II data processing farm. FBS is designed to manage farms composed of up to 1000 computers. It should manage up to 2000 simultaneously running batch user processes. FBS should run arrays (job sections in FBS terms) of concurrent processes. Assuming that typical batch job section will consist of up to 10 processes, and run for up to 10 hours, FBS has to be able to handle more than 200 simultaneously running job sections, and job section start rate from 20 to 200 per hour.

FBS should provide basic job control services such as job submission and cancellation, job scheduling, job status monitoring, basic accounting, resource utilization monitoring.

Since FBS will be used in the environment of large farms of relatively inexpensive and not necessarily too reliable computers, it has to be robust with respect to unexpected shutdown of some number of farm nodes as well as failures of FBS components.

Typical Run II production farm will primarily consist of Linux computers, but will include several large computers running under different operating systems such as IRIX or OSF1. Therefore, FBS should be highly portable across various UNIX platforms.

Due to big number of computers to operate, FBS support and maintenance costs per single computer should be low.

2.2 FBS Design and Features

FBS is designed based on several assumptions.

2.2.1 Farm Model

FBS uses relatively simple model of a farm (Figure 1). Farm consists of computers (worker nodes) of one or more classes (node types). Computers within one class are identical, and FBS can pick any available of them for next user process to start on.

Computers are grouped into classes based on nature and amount of resources available on them. For example, a farm may consist of several multi-processor nodes (CPU nodes), some nodes with additional disk space attached to them (disk nodes), and some number of nodes with relatively fast network connection (network nodes). In this case, one can describe the farm as having 3 classes of nodes. Node classes can be used to logically partition a farm into non-overlapping parts (farmlets).

2.2.2 Load Balancing and Resource Management

One of important FBS' functions is to prevent excessive use of resources and evenly distribute load among farm nodes.

Distinctive characteristic of farm computing is low number of active processes running on each farm node. Therefore, each active process will use substantial amount of farm node resources. In case of 2 or 3 processes per node, each process will be using 50% or 33% of total node resources. Although Run II data analysis processes are expected to be primarily CPU-bound processes, with constant resource utilization, they are going to go through certain (relatively short) phases of relatively low resource consumption, for example, when downloading or uploading data. Such inevitable short term irregularities in individual process resource consumption cause significant irregularities in total node resources utilization. This makes it difficult if not impossible to use simple methods of forecasting future node resources utilization based on current measurements.

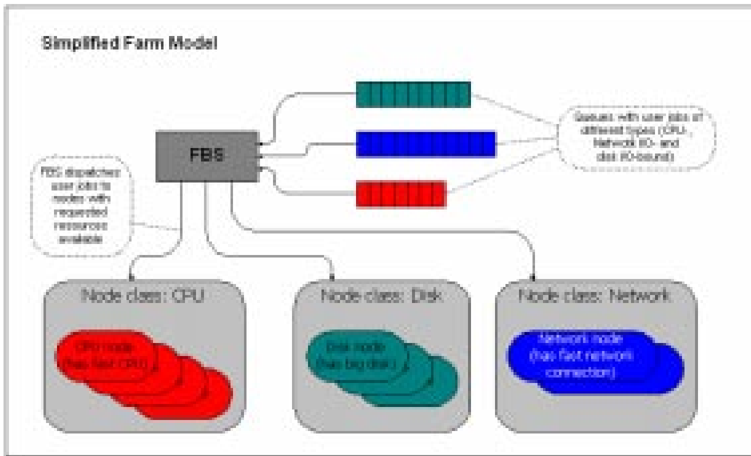

Figure 1: FBS Farm Model

Because FBS is designed for farm architecture, unlike other batch systems, it does not rely on load measurements at all. Instead, FBS uses ”resource counting” method for load balancing. The idea of this method is that FBS administrator has to describe the farm in terms of available resources such as CPU power, disk space, etc. FBS user has to specify resource consumption requirements for each process either at the time of job submission, or as part of FBS configuration. In current design, FBS assumes constant resource utilization level during batch process lifetime, which is very close to the reality for the computational tasks FBS is expected to manage. Based on the knowledge of resource capacity of the farm nodes, resources used by running processes and resource requirements of pending batch jobs, FBS makes the decision when and where to start each batch job.

Because FBS does not actually measure any resources, and knows them only by name and their stated capacity, the resources are referred to as ”abstract resources”. In early versions, FBS was capable of managing only 2 abstract resources: CPU power and disk space on worker nodes. Other types of resources such as magnetic tape drives, network bandwidth, NFS-shared disk space, etc. can be represented as FBS abstract resources.

2.2.3 Major Components

FBS design (Figure 2) consists of the following components:

- FBS User Interface (UI). UI provides command line as well as graphical interface to FBS. UI allows user to submit, monitor and control batch jobs as well as monitor current status and availability of the farm.
- LSF (Load Sharing Facility) is a commercial software package distributed by Platform Software used as a component of FBS. LSF’s responsibilities within FBS are quite limited. It is responsible only for job queuing, scheduling, maintaining persistent job information. LSF does not know anything about farm nodes and does not measure resource availability there. Instead, FBS uses ELIM extension to notify LSF how many more batch processes can be started at any time.
- FLIMD is an FBS daemon that is responsible for keeping track of running processes, resources allocated to running processes, farm nodes status, and reporting in some way resource availability to LSF through ELIM. In addition, FLIMD makes the decision on which farm nodes to start processes of each section.

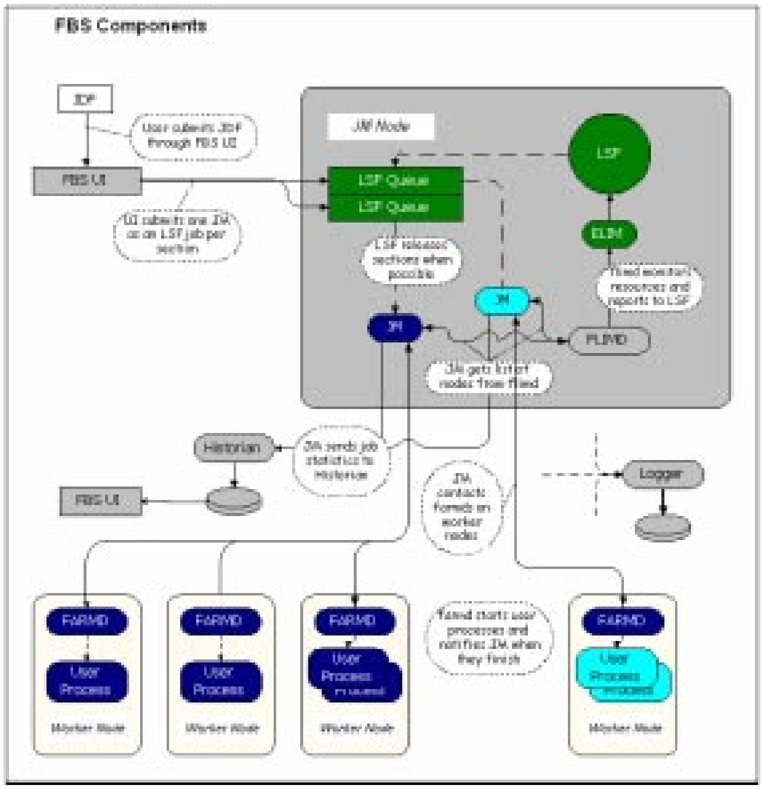

Figure 2: FBS Design

- Job Manager (JM) is an FBS process that controls single section running on the farm. LSF starts JMs as LSF batch processes. JM is responsible for allocating resources on farm nodes with FLIMD, and communicating with FARMD(s) on nodes allocated for the section by FLIMD to start user processes, and wait for their completion.
- FARMD is a FBS daemon that runs on each worker node. It creates environment for user processes, starts them as requested by JM, reports their status to JM and UI, notifies JM when user process exits.
- Historian is FBS historical database manager. It receives section start/exit statistics and stores it on disk. UI provides a tool for reading this database and generating reports as requested by user.
- Logger or log daemon is responsible for receiving and storing error and event log information sent by other FBS components. This information is primarily used for FBS debugging and trouble shooting.

2.2.4 Robustness and Reliability

FBS design makes it highly reliable and robust with respect to failure of individual components. This is achieved by distributing run-time information among different FBS components and avoiding redundancy of the information. Basic idea is that FLIMD, as the most critical component, can recover after failure based on information received from JMs. JM is highly reliable component and most likely reason for its failure is failure of the node where it runs. Since in typical configuration all JMs and LSF run on the same node, failure of the node inevitably means failure of LSF and the whole farm, and necessarily leads to re-initialization of the batch system. Unlikely failure of an individual JM or FARMD leads only to failure affected batch job or process, which is not considered to be dangerous.

2.2.5 Job and Section

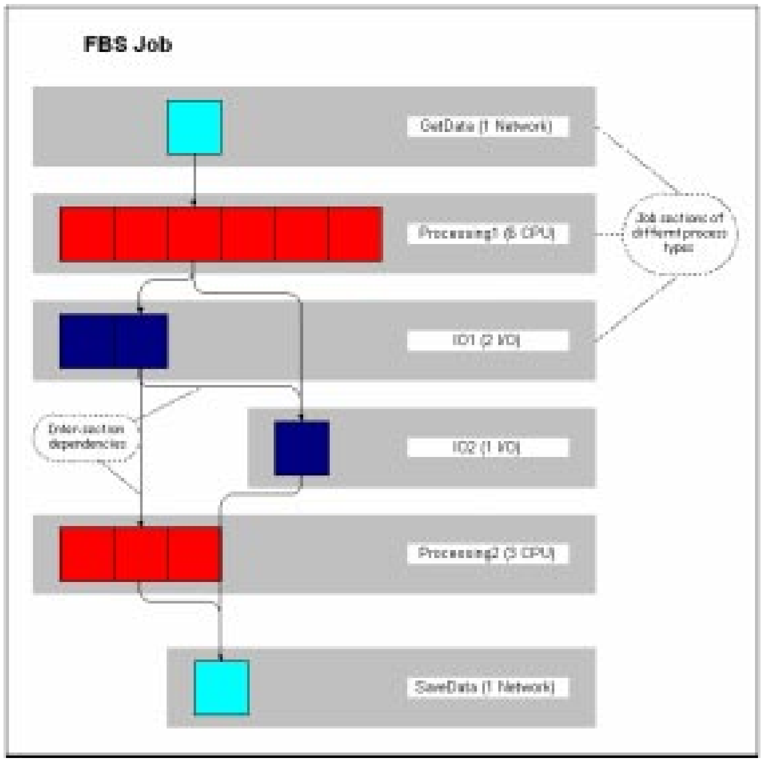

Figure 3: FBS Job Structure

FBS job (Figure 3) consists of one or more sections. Section is an array of parallel processes, running on the same or different nodes of the farm. The processes of the same section are considered to be identical in terms of resources consumption. FBS starts all processes of the same section at the same time. Each section has its unique name within the job.

Although it provides some basic support for users of such parallel computing tools as PVM and MPI, FBS does not have or use any knowledge of how processes of the section communicate with each other or whether they communicate at all.

User can specify dependencies between different sections of the same job. For example, user can request that section A does not start until section B finishes. FBS provides 4 types of dependencies:

- started (start section A only after section B starts)
- done (start section A only when section B finishes successfully)
- exited (start section A only when section B finishes unsuccessfully)
- ended (start section A only when section B finishes regardless of success or failure)

Exit status of a section is calculated based on UNIX exit codes of each individual section process and other section parameters.

2.2.6 Sample Job Description File

The following is an example of FBS Job Description File. This JDF describes 3 sections, Init, Process and CleanUp.

First section is supposed to dump an input tape to disk. It is submitted to the queue named "IO_QUEUE". FBS configuration specifies that sections submitted to this queue consist of I/O-bound processes, and that such processes require relatively low CPU utilization. The section consists of one process. This process requires 3 GB of local disk space on the worker node.

Second section is named "Process". It performs data processing. It is submitted to queue named "CPU_QUEUE". FBS configuration defines resource requirements for processes of sections submitted into this queue according to their CPU-bound nature. The section will start 5 concurrent processes, each will occupy 10 GB of local disk space. The section depends on Init section. Section "Process" will not start until section "Init" finishes successfully. If section "Init" fails, section "Process" will never start.

Last section of this job is supposed to perform clean-up actions in case Process section fails. It will start only if Process section fails.

```
SECTION Init
    QUEUE = IO_QUEUE
    EXEC = my_bin/dump_tape.sh XYZ1234 /mnt/stage/XYZ1234
    NUMPROC = 1
    DISK = 3
SECTION Process
    QUEUE = CPU_QUEUE
    EXEC = my_bin/do_processing.sh /mnt/stage/XYZ1234
    NUMPROC = 5
    DISK = 10
    DEPEND = done(Init)
SECTION CleanUp
    QUEUE = FAST_QUEUE
    EXEC = my_bin/std_cleanup.sh /mnt/stage/XYZ1234
    NUMPROC = 1
    DEPEND = exited(Process)
```

2.3 FBSNG: FBS without LSF

Since the time this article was originally written, FBS project has made significant progress. This effort led to entirely re-designed product named "FBS, Next Generation" (FBSNG). FBSNG [4] inherits the concepts and design ideas of FBS. It does not use LSF as its component any more. This makes FBSNG an independent and complete full-scale batch system for farm architecture.

Major reasons for undertaking the re-design effort were:

- Reducing support, maintenance and licensing costs
- Avoid potential scalability problems caused by the way FBS uses LSF
- FBS design was cumbersome because it had to interact with LSF
- Using LSF as batch job scheduler was a major obstacle in further FBS development and adding necessary features

FBSNG design is shown on Figure 4. First version of FBSNG was released in July 2000, and since then it is being successfully used in production on several PC farms at Fermilab ranging from 37 to 100 nodes, as well as by off-site collaborators. FBSNG successfully passed series of

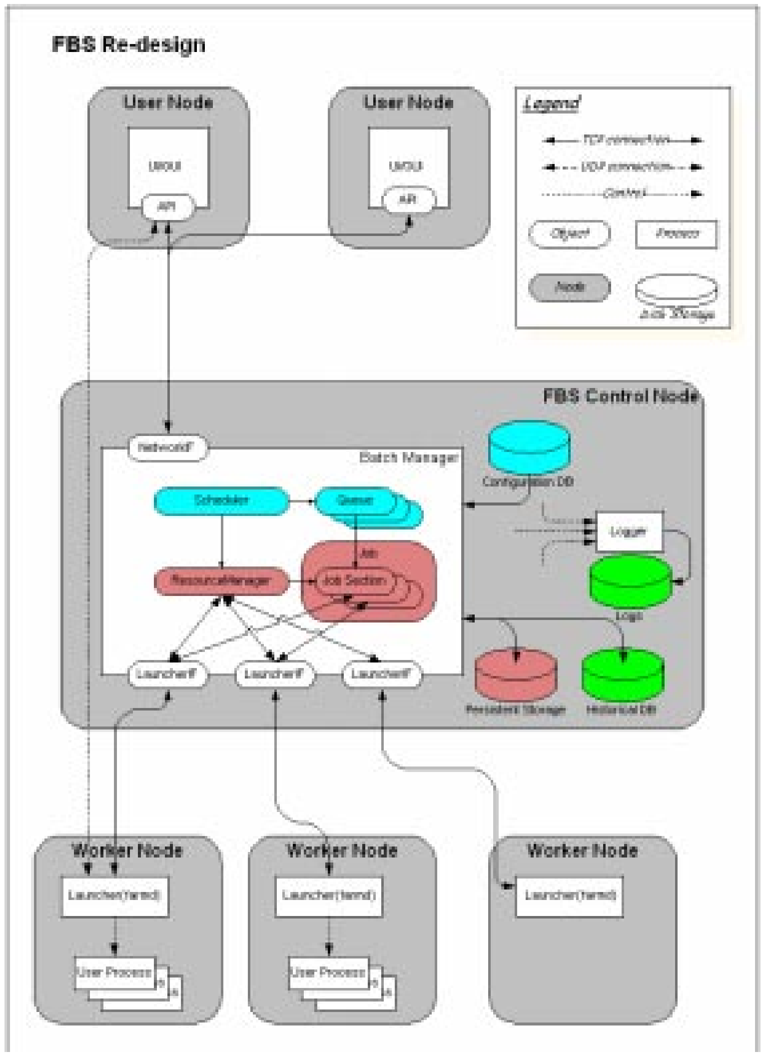

Figure 4: FBS re-design

stress tests designed to determine whether it meets the scalability requirements. These tests have confirmed that FBSNG can successfully handle required number of simultaneously running batch processes and far exceeds job release rate requirements.

Currently, the following additional features are implemented in FBSNG:

- The concept of ”abstract resources” is fully implemented. FBSNG can manage unlimited number of different resources of the following types:
  - Global resources, not associated with any particular farm node such as network bandwidth, NFS-shared disk space;
  - Local resources such as CPU, memory, local disk space, tape drives;
  - Node attributes which can be used to represent such resources as flavor and version of operating system, special software installed. Abstract node attributes such as ”pink” and ”blue” can be and are being used to logically partition the farm into smaller sub-farms or farmlets.
  - Resource pools can be used to describe sets interchangeable resources such as scratch disks available on worker nodes so that FBSNG automatically chooses whichever disk

is available and allocates requested amount of space to the batch process.

- Customizable scheduler designed to manage parallel jobs
- Capability to run a batch job in interactive mode
- Full scale FBSNG Application Programmer's Interface which can be used to build custom data processing systems on top of FBSNG
- Support for Kerberos user authentication

Currently, FBSNG is installed on 5 farms of different sizes at FNAL and several off-site farms and is being successfully used in production.

3 Fermi Inter-Process Communication and Synchronization Toolkit (FIPC)

FBS can be successfully used to manage a computational resource only if its allocation and deallocation is synchronized with the batch process start and finish respectively. Such "synchronized" resources as CPU power, scratch disk space, magnetic tape drives fall into this category.

However, not all computational resources available on a farm behave this way. Some resources are needed to be allocated only for small portions of process lifetime. One example of such "short-term" resource is network bandwidth used to transfer input or output data file over the network. Resources of the other category, "long-term" resources, must remain allocated for indefinitely long time after the batch process finishes, for example, disk space allocated by data produced by the batch process.

FIPC was proposed, designed and developed to help farm users manage allocation of non-synchronized resources and establish communication between cooperating processes running on a farm. Essentially, FIPC is a system that provides users with a set of distributed inter-process communication (IPC) objects accessible from any farm computer.

3.1 FIPC Features

Currently, FIPC provides the following IPC object types:

- Lock is a simple binary semaphore object.
- Gate is a counted semaphore. It resembles an entrance to a room with limited number of seats available.
- Client queue is a first-in-first-out queue which batch processes can enter and wait for an access to certain resources.
- Integer flag is a simple integer variable with a functionality of getting/setting value, waiting until the value of the flag becomes greater than, less than or equal to certain value, and then atomically altering flag's value. Flag can be used as a semaphore.
- General-purpose list has double-ended queue functionality. User can add and remove arbitrary text string items to head or tail of the list. This object can be used as a message queue as illustrated in the example below.
- String variable. It provides basic string operations in terms of the functionality of the standard UNIX Regular Expressions package: getting/setting string value, waiting for value to match some pattern, atomic match-and-set operation.

FIPC provides two level of interface - shell command line interface and API. Currently, only Python binding of FIPC API is available.

FIPC Object names are organized in a name space similar to UNIX file system name space. User can create "(sub)directories" and place FIPC objects in them. The name space as well as all FIPC objects are global: all objects are visible from any node of the FIPC cluster - set of computers where FIPC clients are configured to communicate with the same set of FIPC servers. FIPC objects have ownership and protection attributes. Users can protect their objects from unauthorized access to them.

State altering operations on FIPC objects are atomic: any operation (for example match-and-set operation on a string variable), performed on an FIPC object is guaranteed to finish before the state of the object can be altered by some other operation.

Locks, gates, and client queues provide clean-up functionality. For example, if a client entered into a queue, and then unexpectedly exited without removing itself from the queue, FIPC will clean the queue up removing all non-existing clients.

In order to provide the clean-up functionality, every FIPC client is assigned a unique FIPC ID. There are two types of FIPC clients: single process clients and session clients. Single process client is identified by its UNIX process id. Session client is a UNIX session - group of processes running on the same node with the same UNIX session id. Session client is considered to be alive for the purpose of clean-up functionality as long as there is at least one process of the session running on the client's node. Since every UNIX shell command is executed in a context of a new process with unique process id, and such process inherits session id from its parent shell, session id, but not process id can be used to recognize temporary processes executing individual FIPC commands as related of the same client. Therefore, session clients should be used with FIPC command line interface, whereas FIPC API clients can be single-process clients.

3.2 FIPC Design

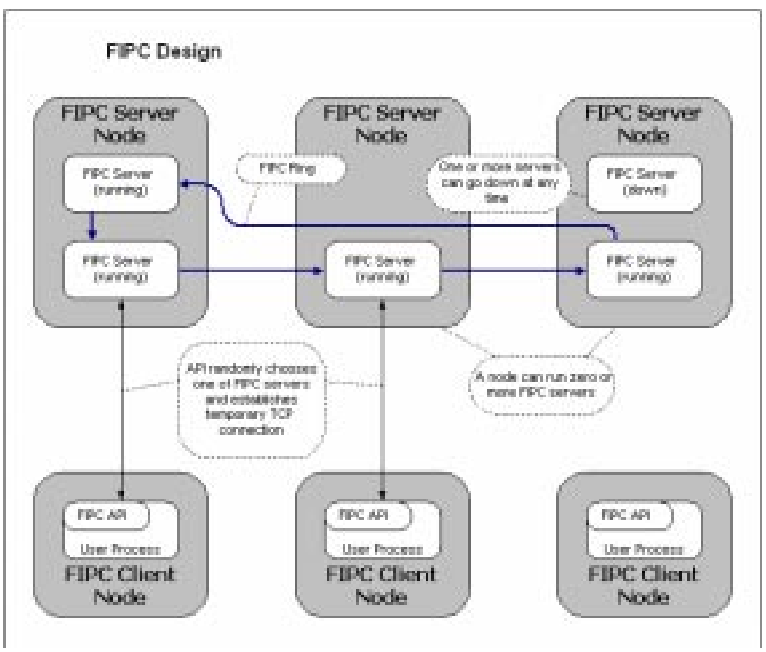

Figure 5: FIPC Design

FIPC consists of the following components (Figure 5):

- FIPC Cluster is a set of nodes where FIPC client software is configured to use the same set of FIPC servers.
- FIPC Servers. There are one or more FIPC Servers running in FIPC cluster. They connect to each other via TCP sockets to form a Ring. Each server connected to the Ring has exactly the same information as any other. This makes FIPC servers completely redundant and al-

lows any server to disconnect from the Ring at any time without causing any loss of information. Servers can re-join the Ring at any time and recover all the information. Usually, FIPC servers are started during system boot time.

- FIPC API is a library that provides access to FIPC servers and IPC objects. FIPC API establishes new connection with randomly chosen available FIPC server for each operation.
- User Interface provides shell-level command line interface to FIPC. It is implemented in terms of FIPC API, and provides the same functionality as API.
- FIPC Locals are simple daemons that are capable to answer whether particular user process or session is still running on the node. This information is used by FIPC to perform clean-up operations on locks, gates and queues.

3.3 Redundancy, Scalability and Robustness

Due to complete redundancy of FIPC servers, FIPC cluster can be built around any number of FIPC servers, but least one server is required to run at any time. FIPC administrator may decide to run more FIPC servers on more than one computer of the FIPC cluster in order to:

- increase robustness of the FIPC with respect to failure of individual FIPC server nodes;
- distribute communication load among several FIPC server processes and computers they are running on;
- allow smooth FIPC upgrades: in most cases, FIPC servers can be shot down, upgraded and re-started one after the other without any loss of information or any other effect on users.

4 FIPC and FBS

The following example illustrates how FIPC and FBS can be used to establish communication and synchronization between cooperating batch processes. The example shows simple solution of the following problem. Suppose a batch job is to generate some number of data files and then process them. The job consists of some number of concurrent ”reader” processes and some number of concurrent ”writer” processes. Readers are guaranteed not to start before writers start. Each writer produces one data file at a time. Each data file has to be processed by one and only one reader process. The following C-shell script and JDF file illustrate how such batch job could be implemented using FIPC and FBS. Instead of generating files of data, in this example, each writer simply generates some message and sends it to a reader. In reality, these messages may contain data file names.

```
#!/bin/csh

# Create set of unique FIPC object names based on FBS job id
set obj_name_prefix = "/rw_test/${FBS_JOB_ID}"
set msg_buf = "${obj_name_prefix}/buf"
set nwr_flag = "${obj_name_prefix}/nwriters"
set nmsg_flag = "${obj_name_prefix}/nmsg_in_buf"

# Create and initialize FIPC objects if necessary
# If the objects already exist, this fragment will do nothing
fipc create list $msg_buf
fipc create flag $nwr_flag 0
fipc create flag $nmsg_flag 0

# Generate unique process ID from FBS-provided variables
set my_id = "${FBS_JOB_ID}.${FBS_SECTION_NAME}.${FBS_PROC_NO}"
```

```
# Get my type, reader, writer, or cleaner
set my_type = "${FBS_SECTION_NAME}"

if ( "$my_type" == "Writer" ) then
    #
    # writer process
    #
    @ nmsgs = $1                # total number of messages to send
    @ buf_capacity = $2         # maximum capacity of message buffer
    fipc fset $nwr_flag + 1     # let readers know we are ready
    @ n = 0
    while ( $n < $nmsgs )
        @ n++
        # Generate message.
        # In reality, generate data file and send its name as a message
        set msg = "Message #${n} from ${my_id}"

        # Allocate room in the buffer by raising semaphore
        fipc fwait $nmsg_flag \< $buf_capacity + 1

        # Put the message in the buffer
        fipc add tail $msg_buf "$msg"
        echo "Sent msg <$msg>"
    end
    fipc fset $nwr_flag - 1
else if ( "$my_type" == "Reader" ) then
    #
    # reader process
    #
    @ nw = 1
    while ( $nw > 0 )

        # Remove first message from the buffer
        set msg = `fipc fetch -t 100 head $msg_buf`
        if ( $status == 100 ) then
            # If timed-out, check if writers are still there
            @ nw = `fipc show flag $nwr_flag`
        else
            # Decrement number of messages in the buffer
            fipc fset $nmsg_buf - 1

            # Process the message
            echo "Received msg <$msg>"
        endif
        #
    end
else
    #
```

```
    # clean-up - destroy FIPC objects
    #
    fipc delete list $msg_buf
    fipc delete flag $nmsg_flag
    fipc delete flag $nwr_flag
endif
```

In the first part of the script, each batch process creates set of FIPC objects. In order to ensure each batch job uses its own set of objects, FBS job id is used to generate job-specific object names. Because FIPC object creation command does nothing when the object already exists, it is safe to issue object creation commands in every batch process. After creating objects, the batch process uses FBS job section name to determine which function, reader, writer, or clean up it is going to perform.

This script uses 3 FIPC objects:

- List object named ”msg_buf” is used as a message queue. Writers add messages to the tail of the list, and readers retrieve them from its head.
- Flag ”nmsg_flag” is a semaphore used to limit queue size. Every time a writer process places a new message in the queue, this flag is incremented, and then reader process decrements it after processing the message.
- Flag ”nwr_flag” is used to signal readers that there are no more active writers in the job, and therefore readers may exit.

Clean-up section of the script destroys FIPC objects used by the job.

```
SECTION Writer
    NUMPROC = 7              # start 7 writers
    EXEC = rw.csh 10 5      # each will generate 10 data files;
                             # limit queue size to 5
    ...
SECTION Reader
    NUMPROC = 5              # 5 readers
    EXEC = rw.csh
    DEPEND = started(Writer)
    ...
SECTION Cleanup
    NUMPROC = 1
    EXEC = rw.csh
    DEPEND = ended(Writer) && ended(Reader)
    ...
```

Notice the use of DEPEND clauses in Reader and Cleanup sections. They make sure that reader processes do not start before writers, and that FIPC objects are destroyed only when they are no longer in use.

Although FIPC was meant to be a package used by FBS users, it does not depend on FBS, and can be used separately in any distributed environment where it is necessary to implement simple communication and/or synchronization between processes. However, if used together, FBS and FIPC form a set of tools, which can be successfully used to build distributed applications in such an environment as off-line data processing farms.

5 Conclusion

FBS and FIPC were designed and developed in Fermilab as elements of software infrastructure for off-line data processing in PC farm environment. Both products have been successfully used for several years on PC farms of different sizes and structures, used by different groups of physicists to perform different types of computing [5],[6]. Both products have proved to be simple yet powerful and flexible tools for farm data processing.

References

1 PBS web page `http://pbs.mrj.com`
2 Platform Computing web page `http://www.platform.com`
3 FBSNG web page `http://www-isd.fnal.gov/fbs`
4 FBSNG web page `http://www-isd.fnal.gov/fbsng`
5 Fermilab preprint Conf-00-095-E
6 Fermilab preprint TM-2109